Chapter 9: Working Environment

This chapter describes the methods used to set up your working environment in UNIX. Some of these are standard UNIX (e.g., shell and environment variables), and some are provided and/or customized by the login scripts provided by FullFUE.

9.1 Special Keys

UNIX has a number of special keys that perform particular functions. Some important ones are the keys necessary to backspace over a character when entering a command, to delete the whole line being entered, and to interrupt execution. These are user-configurable, and have different defaults based on the shell, the version of UNIX, and your login files. If your system has FullFUE installed, run

```
% setup setterm
```

and these special keys will be set as described in the table:

Name	Control Char	Function
erase	DEL or backspace	Erase character. Backspace and erase one character (the key used depends on terminal setting). Sometimes, especially within **tcl/tk** applications, you must use <Ctrl-h>.
werase	<Ctrl-w>	Delete the rightmost word typed in.
kill	<Ctrl-x>	Kill (erase) the line typed in so far. If you prefer to use <Ctrl-u> for this function, uncomment the line `#stty kill '^u'` in your `.login` file (C shell family), or `#stty kill` in your `.profile` file (Bourne shell family).
intr	<Ctrl-c>	Interrupt the program currently running.

Name	Control Char	Function
rprnt	<Ctrl-r>	Reprint the line typed in so far.
flush	<Ctrl-o>	Stops terminal output until you press a key.
susp	<Ctrl-z>	Suspend the program currently running and put it in the background. **This does not stop the process!**
stop	<Ctrl-s>	Stop the display. To resume, press the **start** key (<Ctrl-q>)
start	<Ctrl-q>	Start the display after stop.
eof	<Ctrl-d>	Send the program an end-of-file character.

To display the current settings for your terminal, enter:

`% stty -a`

The output of this command is described in section 9.3 *Terminal Characteristics*. If the keys don't seem to work as described here or you want to change them, refer to that section.

Tabs

UNIX relies on the hardware tabs of your terminal. If they are not set or if they are set in an unusual way, displays may appear strange on your terminal. You can set the tabs manually on your terminal, or you can use the **`tabs`** command to set them. The command with no arguments:

`% tabs`

will set tabs in the usual UNIX way, 8 spaces apart.

Special Note Regarding Backspace and Delete Keys

In some unusual circumstances of setup and keyboards you may also need to issue this set of commands to get the backspace key to work as expected:

`% stty erase "^?"`

`% stty intr "^C"`

`% stty kill "^X"`

Sometimes there is trouble with the delete key. Adding the following text to your `.profile` or `.login` will make the key useful:

```
case $TERM in
vt100)
    stty erase \^? ;;
```

```
xterm)
    case "`xdpyinfo | grep 'vendor string'`" in
        *DigitalEquipmentCorp*)         stty erase \^? ;;
        *Network\ Computing\ Devices*)  stty erase \^H ;;
        *Silicon\ Graphics*)            stty erase \^H ;;
        *)                              stty erase \^H ;;
    esac
    ;;
esac
```

Special Note for ksh Users Regarding Arrow Keys

To make the up and down arrow keys work and therefore to enable command line editing and recall in **ksh**, include the following lines in your `.shrc` file (or `.kshrc`):

```
set -o emacs
alias __A='^P'
alias __D='^B'
alias __B='^N'
alias __C='^F'
```

Note that the `A`, `D`, `B`, and `C` are preceded by *two* underscores, and that you need to insert an actual control character, not simply carat-P or carat-B. A control character typically needs to be preceded by a "quoting" character, which differs from editor to editor.

For this (these) editor(s):	... enter this immediately before the control character:
vi	<Ctrl-v>
emacs	<Ctrl-q>
NEdit	Use Insert Control Character from the Edit menu.

We believe this prescription works on all UNIX operating systems, regardless of how you're connected (e.g., **telnet**, **xterm**).

9.2 Special Characters (Metacharacters)

9.2.1 Slashes

Backslash

The backslash (****) character is used on the UNIX command line to mask the special meaning of the character immediately following it (no spaces inbetween) so that the command interpreter takes the character literally. It is called a *quoting* character. For example:

```
% <command> \<CR>
```

causes the carriage return (`<CR>`) to be ignored, allowing you to continue typing your command on the following line.

Forward Slash

The forward slash (**/**) character is the symbol for the `root` directory. In path names it acts as a separator between directories in the hierarchy, and between the last directory and the file, if one is specified. For example:

```
/<dir>/<subdir_1>/.../<subdir_n>/filename
```

9.2.2 Quotes and Parentheses

Different types of quotes have special meanings:

- Normal single quotes (apostrophes) around a string (`'string'`) tell the command interpreter to take the string (`string`) literally.
- Double quotes around a string (`"string"`) also tell the command interpreter to take the string literally, but allow interpretation of variables that follow a **$** character (**$** preceding a variable name outputs the value of the variable; see section 9.5 *Shell Variables and Environment Variables*).

  Section 6.1.2 *Command Interpretation by the Shell* further explains single and double quotes in command interpretation, and provides an example.
- Single backquotes around a command string (`` `string` ``) tell the interpreter to run the command(s) in the string, and to use the output of the command(s) in place of the string itself. This is useful for combining two commands into one, and for doing iterative tasks within shell scripts (shell scripts are introduced in section 5.4 *Shell Scripts*).

- A string of commands enclosed in parentheses (e.g., **`(<command1>;<command2>)`**) is run in a subshell. (In section 6.1.1 *Programs, Commands and Processes* we discuss the difference between shell commands and non-shell commands. A non-shell command always runs in a subshell; when enclosed in parentheses, the command starts a second subshell.)

Command Separators

- The semicolon (**`;`**) character separates successive commands on a single command line. For example,

  `% <command1> ; <command2>`

  executes **`<command1>`**, and when it finishes, **`<command2>`** gets executed.
- The ampersand character (**`&`**) is similar to the semicolon (**`;`**) but does not wait for **`<command1>`** to finish.
- A double ampersand (**`&&`**) runs **`<command2>`** only if **`<command1>`** was successful.
- Piping commands is discussed in section 6.4.3 *Pipes*. A pipe (the pipe symbol **`|`**) tells **`<command2>`** to use the output of **`<command1>`** as input.
- A double pipe (**`||`**) runs **`<command2>`** only if **`<command1>`** fails.

Other Special Characters

Special characters such as the asterisk (**`*`**), the question mark (**`?`**), square brackets (**`[...]`**) are used as wildcards in file expansion (section 7.2.2 *Filename Expansion and Wildcard Characters*). Note that to prevent file expansion, these characters must be prefaced by a backslash (**`\`**).

Other sets of characters are used in input/output redirection (redirection metacharacters, see section 6.4.2 *Standard Input and Output Redirection*), and in regular expressions (section 6.4.5 *Regular Expressions*) as wildcards, delimiters, and other special pattern-matching characters. Refer to these sections for specific information.

9.3 Terminal Characteristics

You can specify your terminal type to UNIX if the default is not suitable. To do so, enter the command for the C shell family:

```
% set term=<termtype>
```

or for the Bourne shell family:

```
$ TERM=<termtype>; export TERM
```

where **<termtype>** is the name of a terminal type supported on the system. vt100, vt220 and xterms are acceptable terminal types. If you always use the same kind of terminal, you may want to put this command in your `.login` or `.profile`. Note that the standard Fermi files attempt to set this variable correctly.

In Section 9.1 *Special Keys* we listed some terminal control functions. Recall that you can display the settings with the **stty** command:

```
% stty -a
```

The format on each machine is different but should indicate approximately the same information. The following is the output from a Silicon Graphics workstation. The settings reflect the FUE defaults.

```
speed 9600 baud; line = 1;
intr = ^C; quit = ^; erase = DEL; kill = ^X; eof = ^D; eol
= ^@; swtch = ^Z
lnext = ^V; werase = ^W; rprnt = ^R; flush = ^O; stop = ^S;
start = ^Q
-parenb  -parodd  cs8  -cstopb  hupcl  cread  clocal  -loblk
-tostop
-ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr
icrnl -iuclc
ixon ixany -ixoff
isig icanon -xcase echo echoe echok -echonl -noflsh
opost  -olcuc  onlcr  -ocrnl  -onocr  -onlret  -ofill  -ofdel
tab3
%
```

In this display the second and third lines display the FUE default control characters. The character **^** indicates the control key (e.g., **^C** represents <CTRL-C>). Your reference books will most likely tell you to delete a character with the # key and delete a line with the @ key, **but this is not correct under FUE**. Use the character indicated as **ERASE** in the **stty** output for single character deletion, and **kill** for whole line deletion. The Fermi UNIX Environment defaults for these operations are the **DELETE** key and **<CTRL-X>**, respectively.

You can display a description of all of the options reported by **stty** with the command:

```
% man stty
```

If you don't like the FUE defaults, you can also set these functions with the **stty** command. The form for setting them is:

```
% stty <control-char> <c>
```

where:

`<control-char>`	is one of the functions in the table in section 9.1 *Special Keys*.
`<c>`	is the representation of the key to be used for that function. A control character is specified preceded by a caret: **`^x`** represents **<CTRL-X>**.

Example:

```
% stty kill '^y'
```

There are two special representations: **`^?`** is interpreted as the **DELETE** key and **`^-`** is interpreted as undefined. You must include the quotes as shown in the example so that special characters are not interpreted incorrectly. You must be careful not to have two functions represented by the same key.

There are many other options that can be set with **stty**. Others that might be of interest are **`echoe`** which specifies that deleted characters are erased, and **`-tabs`** which specifies that the tab character be translated into the appropriate number of spaces. Refer to the man pages for more information.

9.4 Information Distribution System: NIS

NIS (Network Information System) is a system that distributes information throughout a cluster. We define a UNIX cluster as a group of machines that share both a common password file (or user database), and a common file system, especially for login directories. NIS is usually used to provide the common password file, and the common file system is typically NFS or AFS.

NIS is installed on FNALU and many other UNIX clusters at Fermilab. In order to determine if NIS is running on your system, execute the command:

```
% domainname
```

If it returns a value, then NIS is running on your cluster. If no output is returned, then it is not. Many UNIX clusters use NIS to share a common login area across several machines. Note that it is possible for both AFS and NIS to be installed on a system.

9.5 Shell Variables and Environment Variables

Every UNIX process runs in a specific *environment*. An environment consists of a table of *environment variables*, each with an assigned value. When you log in certain *login files* are executed. They initialize the table holding the environment variables for the process. (Exactly which files run will be made clear later in this chapter.) When this file passes the process to the shell, the table becomes accessible to the shell. When a (parent) process starts up a child process, the child process is given a copy of the parent process' table. Environment variable names are generally given in upper case.

The shell maintains a set of internal variables known as *shell variables*. These variables cause the shell to work in a particular way. Shell variables are local to the shell in which they are defined; they are not available to the parent or child shells. Shell variable names are generally given in lower case in the C shell family and upper case in the Bourne shell family.

9.5.1 C Shell Family

The C shell family explicitly distinguishes between *shell variables* and *environment variables*.

Shell Variables

A shell variable is defined by the **`set`** command and deleted by the **`unset`** command. The main purpose of your `.cshrc` file (discussed later in this chapter) is to define such variables for each process. To define a new variable or change the value of one that is already defined, enter:

```
% set <name>=<value>
```

where **`<name>`** is the variable name, and **`<value>`** is a character string that is the value of the variable. If **`<value>`** is a list of text strings, use parentheses around the list when defining the variable, e.g.,

```
% set name=(<value1> <value2> <value3>)
```

The **`set`** command issued without arguments will display all your shell variables. You cannot check the value of a particular variable by using **`set <name>`**, omitting **`=<value>`** in the command; this will effectively unset the variable.

To delete, or unset, a shell variable, enter:

```
% unset <name>
```

To use a shell variable in a command, preface it with a dollar sign (**$**), for example **$<name>**. This tells the command interpreter that you want the variable's value, not its name, to be used. You can also use **${<name>}**, which avoids confusion when concatenated with text.

To see the value of a single variable, use the **echo** command:

```
% echo $<name>
```

If the value is a list, to see the value of the *n*th string in the list enter:

```
% echo $<name>[<n>]
```

The square brackets are required, and there is no space between the name and the opening bracket.

To prepend or append a value to an existing shell variable, use the following syntax:

```
% set name=prepend_value${name}
```

or

```
% set <name>=${<name>}<append_value>
```

Note that when a shell is started up, four important shell variables are automatically initialized to contain the same values as the corresponding environment variables. These are *user*, *term*, *home* and *path*. If any of these are changed, the corresponding environment variables will also be changed.

Environment Variables

Environment variables are set by the **setenv** command, and displayed by the **printenv** or **env** commands, or by the **echo** command as individual shell variables are. Some environment variables are set by default (e.g., *HOME*, *PATH*).

The formats of the commands are (note the difference between **set** and **setenv**):

```
% setenv [<NAME> <value>]
```

```
% unsetenv <NAME>
```

where **<value>** is interpreted as a character string. If the string includes blanks (i.e., if it encompasses multiple values), enclose the string in double quotes (**"**), e.g.,

```
% setenv NAME "<value1> <value2> ..."
```

The current environment variable settings can be displayed using the **setenv** command with no arguments.

To use an environment variable in a command, preface it with a dollar sign (**$**), for example **$***NAME*. This tells the command interpreter that you want the variable's value, not its name, to be used. You can also use **${***NAME***}**, which avoids confusion when concatenated with text.

To prepend or append a value to an existing environment variable, use the following syntax:

```
% setenv <NAME> "<prepend_value>${<NAME>}"
```

or

```
% setenv <NAME> "${<NAME>}<append_value>"
```

If the pre- or appended value is the value of a preexisting environment variable, enclose the variable name in braces, too, e.g.,

```
% setenv <NAME> "${<NAME>}${XYZ_VAR}"
```

Appending and prepending is commonly used with the *PATH* variable, and a colon is used as a separator, e.g.,

```
% setenv PATH "${PATH}:${XYZ_DIR}"
```

9.5.2 Bourne Shell Family

The Bourne shell family does not really distinguish between shell and environment variables. When a shell starts up, it reads the information in the table of environment variables, defines itself a shell variable for each one, using the same name (also uppercase by convention), and copies the values. From that point on, the shell only refers to its shell variables. If a change is made to a shell variable, it must be explicitly "exported" to the corresponding environment variable in order for any forked subprocesses to see the change. Recall that shell variables are local to the shell in which they were defined.

Shell variables are defined by assignment statements and are unset by the **unset** command. The format of the assignment statement is:

```
$ NAME=<value>[; export <NAME>]
```
[1]

where **<NAME>** is the variable name, and **<value>** is a character string that is the value of the variable. There are no spaces around the equal sign (**=**). The **unset** command format is:

```
$ unset <NAME>
```

If the string includes blanks (i.e., if it encompasses multiple values), enclose the string in double quotes, e.g.,

```
$ NAME="<value1> <value2> ..."
```

1. In most cases you will want to include the optional part of the command, so that it reads: *NAME***=***value***; export** *NAME*

The values of all the current variables may be displayed with the **set** command.

To use a variable in a command, preface it with a dollar sign (**$**). This tells the command interpreter that you want the variable's value, not its name, to be used. For example, to see the value of a single variable, enter:

```
$ echo $<NAME>
```

You can also use **${<NAME>}**, which avoids confusion when concatenated with text.

To prepend or append a value to an existing environment variable, use the following syntax:

```
$ <NAME>=<prepend_value>$<NAME>
```

or

```
$ <NAME>=$<NAME><append_value>
```

Appending and prepending is commonly used with the *PATH* variable, and a colon is used as a separator, e.g.,

```
$ PATH=${PATH}:${XYZ_DIR}
```

9.6 Some Important Variables

These variables are important for all shells, unless noted otherwise.

DISPLAY

In order to use an X windows application, the environment variable *$DISPLAY* must be set correctly. Normally the FullFUE login files set it correctly for you. Its value is of the form **<node:screen.server>**. At Fermilab, this will generally look like **<node>.fnal.gov:0.0** where **<node>** is your machine name.

To find the node name on a UNIX workstation, run **funame -n**.

HOME

Your home directory is the top of your personal branch in the file system, and is usually designated by your username, i.e., /<path>/<username>. The value of the variable *HOME* is the pathname of your home directory. The command **cd** without arguments always returns you to $HOME. In all shells except **sh**, the tilde (**~**) symbol used in filename expansion, expands to the value of this variable. For example `~/myfile` is equivalent to `$HOME/myfile`. The structure `~<username>` is equivalent to the $HOME directory of user `<username>`.

PATH

The *PATH* variable lists the set of directories in which the shell looks to find the commands that you enter on the command line. (For the C shell family, the shell variable *path* takes its value from *PATH*.) If the path is set incorrectly, some commands may not be found. If you enter a command with a relative or absolute pathname, the shell will only search that pathname for it, and not refer to *PATH*.

If you include the current working directory, "dot" (`.`), in your *PATH*, the shell will always find your current working directory. This allows you to run executable files from your current working directory by typing in only the filename. The FullFUE login files include the dot at the end of the path for you.

For the C shell family, see the following line in the `setpath.csh` file:

```
set path = ($path .)
```

For the Bourne shell family, see the following line in the `setpath.sh` file:

```
PATH = ${PATH}.:
```

See section 9.8 *Tailoring Your Environment* for information on these files. If "dot" is not in your *PATH*, then in order to execute a file, you need to precede the executable filename by `./` on the command line. This provides the current directory pathname explicitly.

LINES and COLUMNS

These variables control the number of lines and columns are displayed on your screen. The **csh** family syntax is:

```
% setenv LINES <n>
% setenv COLUMNS <n>
```

to set the number of lines or columns to **`<n>`**.

☞ For Solaris, use instead:

```
% stty -rows <n>
% stty -cols <n>
```

MANPATH

The *MANPATH* variable lists the set of directories in which the shell looks to find man pages.

SHELL

This variable is set to your default shell. Your default shell is determined by the last field in your password entry (see section 5.1.2 *Starting a Shell*).

ignoreeof

This shell variable is in **csh**, **tcsh**, **ksh** and **bash**, but not in **sh**. When the *ignoreeof* variable is set, you cannot exit from the shell using **<CTRL-D>**, so you cannot accidentally log out. You must use **`exit`** or **`logout`** to leave a shell (see section 3.6 *Logging Out*).

noclobber

This shell variable is in **csh**, **tcsh**, **ksh** and **bash**, but not in **sh**. With the *noclobber* variable set, you are prevented from accidentally overwriting a file when you redirect output. It also prevents you from creating a file when you attempt to append output to a nonexistent file.

noclobber has no effect on utilities such as **cp** and **mv**. It is only useful for redirection. See sections 6.4.2 *Standard Input and Output Redirection* and 7.3 *Manipulating Files*.

9.7 The Alias Command

The **`alias`** command allows you to create your own names or abbreviations for commands by performing string substitution on the command line according to your specifications. Aliases are recognized only by the shell that invokes them; spawned processes do not "inherit" them.

Never use the actual command syntax as an alias for itself. If for some reason an error occurs and the login file which defines your aliases doesn't run, UNIX executes the standard version of the command. Normally you'd see an error message in this case, but what if you miss it? This can be disastrous. For example, if you are accustomed to using **`rm`** (*remove file(s)*, see section 7.3.6 *Remove a File: rm*) as an alias for **`rm -i`** (*remove file(s), but prompt for confirmation*), when you run **`rm`** you will expect a confirmation prompt. If the alias didn't get defined you won't get a prompt, and you may end up removing files you need. That is why we suggest **`rmi`** as an alias for this command.

9.7.1 C Shell Family

The format of the **`alias`** command is:

% alias [<new> [<old>]]

When you enter **<new>** the shell substitutes **<old>**.

The first example causes **ls -l** to be executed when the command **ll** is entered:

% alias ll ls -l

The next example creates the command **dir** to list directory files only:

% alias dir 'ls -l | grep ^d'

grep in this case searches for a **d** in the first column of each line.

9.7.2 Bourne Shell Family

Alias

The **alias** command is supported by **ksh** and **bash**, but not **sh**. For the entire Bourne shell family you can use *shell functions* instead of aliases; we discuss these below. The format of the **alias** command is:

% alias <name> = '<alias_contents>'

The first example causes **ls -l** to be executed when the command **ll** is entered:

% alias ll='ls -l'

The next example creates the command **dir** to list directory files only:

% alias dir='ls -l | grep ^d'

grep in this case searches for a **d** in the first column of each line.

Shell Functions

The Bourne and Korn shells support shell functions, which are similar to shell scripts in that they store a series of commands for execution at a later time. Shell functions are more quickly accessed than scripts because they are stored in memory instead of a file, and the shell preprocesses them. They can be used in place of aliases in order to be completely portable between **sh**, **ksh**, and **bash**.

The format for declaring a shell function is:

```
function-name()
{
   commands
}
```

where `function-name` is what you'll use on the command line to call the function. Typically people declare functions in their `.profile`. You can include anything you'd include in a shell script. For more information on shell functions, see a UNIX text.

9.8 Tailoring Your Environment

This section discusses the FUE-customized login files (also called the *Fermi files*) used to set up your UNIX environment. Under FullFUE you will automatically have your own copy of these files in your home directory[1]. The default files exist in `/usr/local/etc` and you can recopy them to your home directory if you ever need to. Once you understand the functions of the various files, you can tailor them to suit your tastes.

Many of these files include sample code that you may want to activate. A pound sign (#) in the first column indicates a comment line. To activate a command line that's been "commented out", remove the `#`.

9.8.1 C Shell Family Fermi Files

The C shell executes hidden FUE-customized files at various times in your session. They include the files `.cshrc` and `.login`, which you may choose to further modify.

When you log out, the shell looks for a logout script in your home directory called `.logout`. FullFUE does not provide this file, but you can create it yourself, and it will get run automatically. This file is not required.

As an example, including the **`clear`** command in your `.logout` file contents clears the screen when you logout.

We also recommend including the **`kdestroy`** command in the `.logout` file to clear your Kerberos credentials and your AFS tokens, if any.

.cshrc and fermi.cshrc

Upon logging in, the first file to execute is the `.cshrc` located in your home directory. The shell also executes this file each time you invoke a new C shell, for example when you execute a C shell script or otherwise fork a new process.

Your `.cshrc` file:

- sets up the machine id, type, and operating system

1. Exceptions are the `fermi.*` and `setup.*` files which are called directly from `/usr/local/etc`.

- sets up **UPS**
- establishes a reasonable default *path* (and therefore *PATH*) by running `/usr/local/etc/setpath.csh`, and *MANPATH*
- sets *fermimail* as the standard mail alias
- runs **`setup shrc`**

`fermi.cshrc` also calls `/usr/local/etc/local.cshrc` which may set other environment variables.[1] You do not have a copy of `fermi.cshrc` in your home directory; it is not designed to require individual customization.

The file `.cshrc` should contain all your aliases so that child processes have access to them; many suggested aliases are provided for you to activate, and you can define your own. You can also set shell variables (*noclobber* and *ignoreeof* are already set for you) and parameters that are local to a shell.

Don't set any environment variables here. Any changes to their values will remain after you terminate a forked process, thus changing your standard environment for the duration of your login session.

.login and fermi.login

The `.login` file is executed only at login time. After execution of `.cshrc`, the `.login` file located in your home directory is run. The default `.login` file first executes the file `/usr/local/etc/fermi.login`.

`fermi.login` performs several actions:

- sets **umask** (default file access permissions) so others can read and execute but not modify or delete your files
- determines the terminal type (and makes "best effort" at determining *DISPLAY* variable)
- sets common terminal characteristics
- sets a host of environment variables

You do not have a copy of `fermi.login` in your home directory; it is not designed to require individual customization.

Next, the `.login` file sets your prompt, and sets the variables *history* and *savehist*. You can edit your `.login` to modify your path and/or terminal settings, change the default values of environment variables or create your own, and/or include commands that you want to execute once, at the beginning of each session (for instance **`setup <product>`** commands).

1. This is a file for things that the local system manager wants to add to the login scripts. It may or may not have been created on your system.

.logout

The C shell executes the `.logout` file in your home directory (if you have created one) when you log off the system.

Execute files to modify current session

If you modify your `.cshrc` or `.login` files and you want them to take effect in the current session, you must execute them with the **`source`** command:

`% source .cshrc`

`% source .login`

This is explained in section 5.4 *Shell Scripts*.

9.8.2 Bourne Shell Family Fermi Files

The Bourne shell executes hidden FUE-customized files at various times in your session. When you log on in the Fermi environment, the `.profile` and `.shrc` files in your home directory are executed for **sh**, **bash**, and **ksh**. Your `.shrc` file is also executed at any time a new **bash** or **ksh** is invoked.[1]

The name of the file `.shrc` is determined by the *ENV* environment variable which is set to `~/.shrc` in the standard `.profile`, it is not a standard UNIX feature.

.profile and fermi.profile

The `.profile` file first executes `/usr/local/bin/fermi.profile`. This file performs several actions:

- sets **umask** (default file access permissions) so others can read and execute but not modify or delete your files
- sets up the machine id, type, and operating system
- establishes *PATH* (by running `/usr/local/etc/setpath.sh`) and *MANPATH*.
- determines the terminal type
- sets a host of environment variables
- sets common terminal characteristics

1. On some of the more recent OS releases `/bin/sh` is a link (links are described in section 7.3.5 *Reference a file: ln*) to the korn shell (**ksh**). **ksh** is a superset of **sh**, so this shouldn't present any problems for you. One difference is that your `.shrc` file gets sourced when you run `/bin/sh` scripts.

You do not have a copy of `fermi.profile` in your home directory; it is not designed to require individual customization.

The `.profile` file sets your prompt and the variables that govern your history list, your default editor, and your command line editor. You can edit your `.profile` to modify your path and/or terminal settings, change the default values of variables[1] or create your own, and/or include commands that you want to execute once, at the beginning of each session (for instance **`setup <product>`** commands).

.shrc and fermi.shrc

The `.shrc` file first executes `/usr/local/etc/fermi.shrc` which sets up **UPS** and performs some machine-dependent functions.

The `.shrc` file should contain all your aliases[2] so that child processes have access to them; many suggested aliases are provided for you to activate, and you can define your own. You can also set variables (*noclobber* and *ignoreeof* are already set for you except in **sh**) and parameters that are local to a shell, and you can activate and define functions.

Execute files to modify current session

If you modify your `.shrc` or `.profile` files and you want them to take effect in the current session, you must execute them with the **`.`** command:

```
$ . .shrc
$ . .profile
```

This is explained in section 5.4 *Shell Scripts*.

9.8.3 Storing Customized Code

If you wish to maintain versions of distributed code customized to your own needs, we recommend that you store them in the following directories:

`$HOME/bin`	for machine-neutral code
`$HOME/bin.$ARCH`	for architecture-specific code; `$ARCH` is the value returned by **`funame -s`** (e.g., SunOS, IRIX).

The path names for these directories will be added to your *PATH* when the Fermi files are invoked.

1. Remember for **sh**, there is not really a difference between shell and environment variables; see section 9.5.
2. Aliases are available for **bash** and **ksh**, but not for **sh**; see section 9.7.2.

9.9 Multimedia File Support

Applications such as Web browsers and mail handlers need to be able to handle files of many different types. The standard used for identifying multimedia file types is called Multipurpose Internet Mail Extensions (MIME).

When a server sends a document to a client, it usually includes a section that identifies the document's type so that the file can be presented properly. The identifier is called a MIME type, and consists of a general type (e.g., text, image, application, audio, video) and a subtype which specifies the format. These two elements are separated by a slash. Examples of MIME types are:

```
text/plain
text/html
image/jpeg
image/gif
application/postscript
```

A file called `.mailcap`[1] is used to map MIME types to external viewer programs, thus providing a recipe for displaying/playing multimedia files. When you first run **`setup www`** (or **`setup <your-favorite-browser>`** in a FUE environment, a default `.mailcap` file gets created in your `$HOME` directory if it doesn't already exist. If an earlier version of the file is found, the terminal displays a message saying that you can update it from the file in `$NETSCAPE_DIR/lib/TypeMap`. For most situations, the `.mailcap` file should be sufficient as provided.

Each entry in the `.mailcap` file consists of two fields separated by a semicolon (`;`). The first field is the MIME type in the format `type/subtype`. You may see asterisks used as wildcards to specify all of the subtypes of a particular type (e.g., `video/*`). The second field specifies the display command. It requires a full shell command, including the pathname for the external viewer and any command line arguments. Some examples of entries from the `TypeMap` file are:

```
image/xwd; display %s
image/x-xwd; display %s
image/x-xwindowdump; display %s
audio/*; sfplay %s
video/mpeg; mpeg_play %s
video/*; animate %s
application/postscript; ghostview %s
application/x-dvi; xdvi %s
application/pdf; xpdf %s
```

1. Its name refers to the fact that it was originally designed for multimedia mail, however its role has since expanded, and will probably continue to do so as more and more programs become multimedia.

The `%s` is a **printf**-style parameter (see **`man printf`**) for the string representing the filename.

Sometimes, if the MIME type is not sent in the file's header, the multimedia application displaying it needs to determine the file's MIME type from its file extension. In this case, the application references a file called `mime.types` which provides the mapping between file extensions and MIME types. This file is usually not required, and in fact a default `mime.types` file is not even provided.

To add support for a new MIME type with an associated file extension, you would need to create a `mime.types` file to provide the file extension mapping, and then edit your `.mailcap` file to include an entry that maps the new MIME type to an external viewer that can display the data.

For example, say you want to add support for MIME type `application` with (fictional) subtype `xyz`. The files come with the extension `xyzz`, viewable via the program **viewxyz**. You would need to create `mime.types` and include the following line in it:

```
application/xyz xyzz
```

Note there is no semicolon (`;`) in a `.mime.types` entry. Then in `.mailcap`, you would need an entry as follows:

```
application/xyz; viewxyz %s
```